

**Appendix.**

```

#define NAP_CMD_LOGIN_LENGTH      128    /* bytes */
#define NAP_CMD_NEWADDR_LENGTH    64     /* (username + 17) bytes */

/* These servers are the responses to a DNS lookup of "server.napster.com" */
5  #define NAP_NUM_SERVERS      6
    static int sNapServerTable[NAP_NUM_SERVERS] = {
        0x407c2910, /* IP 64.124.41.16 */
        0x407c2911, /* IP 64.124.41.17 */
        0x407c2912, /* IP 64.124.41.18 */
10     0x407c2913, /* IP 64.124.41.19 */
        0xd0b8d8de, /* IP 208.184.216.222 */
        0xd0b8d8df, /* IP 208.184.216.223 */
    };

15  /* These are commonly used Napster port numbers */
    #define NAP_NUM_INIT_PORTS 1
    static int sNapInitPortTable[NAP_NUM_INIT_PORTS] = { 8875 };
    #define NAP_NUM_CMD_PORTS 1
    static int sNapCmdPortTable[NAP_NUM_CMD_PORTS] = { 7777 };

20  /* Note: the services table already has an entry for 8888 */
    /* Note that Napster 2.0 Beta 5 also used ports 4444, 5555, and 6666 */
    #define NAP_NUM_DATA_PORTS 2
    static int sNapDataPortTable[NAP_NUM_DATA_PORTS] = { 6688, 6700 };

25  /* Note: the services table already has an entry for 6699 */
    napFirstLook() {

        /* Temp flags */
30     BOOL  probablyNapInit = FALSE;
        char*dataStart; /* The start of the payload data unit */
        UINT16 cmdLength;
        UINT16 cmdType;
        struct napFlowListData curFlow;
35     struct napFlowListData *savedFlow;
        NessFlowPtr nf;
        SIDE      side1, side2;
        int      port2;
        int      addr2;
40     int      i;

        /* If this is a TCP flow then check to see if it is this Application */
        if (ipf->tcb) {

45         /* Set shortcuts */
            dataStart = ipf->header.dataStart;
            nf = nessFlowFromIPF(ipf);
            side1 = bcb_direction(ipf->bcb);
            side2 = (side1 ? 0 : 1);
50         port2 = nf->conn.port[side2];
            addr2 = nf->conn.addr[side2];

            /* ----- Look for connections to server.napster.com ----- */

55         if (rsp == 1) {

```

```

5      /* Check known port numbers */
      for (i=0; i<NAP_NUM_INIT_PORTS; i++) {
          if (port2 == sNapInitPortTable[i]) {
              probablyNapInit = TRUE ;
              break ;
          }
      }

10     /* Check known addresses */
      if (!probablyNapInit) {
          for (i=0; i<NAP_NUM_SERVERS; i++) {
              if (addr2 == sNapServerTable[i]) {
                  probablyNapInit = TRUE ;
                  break ;
              }
          }
      }

15     if (probablyNapInit) {
        debugBlurt("Detected Napster Server");

        /* Try to dig out the redirection */
        napParseInitialConnection( ipf->header.dataLen, dataStart);
        return SVC_NAP_INIT ;
    }

20     /* Inspect the first 5 packets for Napster, then give up */
    if ( (req + rsp) <= 5 ) {

30         /* ----- Look for Napster Login ----- */

        /* This is a safety net in case we missed the Init traffic */
        /* Anything on these ports will be classified as Napster-Cmd
        35         * even though they are not registered to Napster. */

        for (i=0; i<NAP_NUM_CMD_PORTS; i++) {
            if (port2 == sNapCmdPortTable[i]) {
                return SVC_NAP_CMD ;
            }
        }

40         /* Otherwise inspect the flow for things that look like
        Napster commands */

        if (ipf->header.dataLen >= 4) {

            /* If the first two bytes contain the length of the packet
            minus 4 bytes, then check for a login 'type' value */
            /* Note: The fields are not in network byte order, they are
            50             in little-endian form. */

            cmdLength = (UINT16)((dataStart[1] << 8) + dataStart[0]);

            if (cmdLength == (ipf->header.dataLen - 4) ) {

55                 /* Check for a known login 'type' */

```

```

cmdType = (UINT16)((dataStart[3] << 8) + dataStart[2]);

/* Currently there are two login messages, types 2 and 6 */
if (((cmdType == 2) || (cmdType == 6)) &&
5   (napParseCmdLogin(ipf, cmdLength, (dataStart+4)))) {
    debugBlurt("Detected a Napster-ish login ");
    return SVC_NAP_CMD ;
}
}
10   }

/* ----- Look for raw Napster Data flows ----- */

/*
15   * Anything on these ports will be classified as Napster-Data
   * even though they are not registered to Napster.
   */
for (i=0; i<NAP_NUM_DATA_PORTS; i++) {
    if (port2 == sNapDataPortTable[i]) {
20       return SVC_NAP_DATA ;
    }
}

/* Otherwise inspect the flow for things that look like
25   Napster downloads or uploads */
/* If data length is 1 byte and that byte is the ASCII "1"
   then check the response in the next packet */
if ((ipf->header.dataLen == 1) &&
30   (*ipf->header.dataStart == '1')) {

    /* Save this flow in a list of potential Napster flows */
    if (!(savedFlow =
        (struct napFlowListData *)kmallocc(
            sizeof(struct napFlowListData), M_NESS))) {
35       info0("Unable to save Napster-Data info");

        /* Bail out because we're out of memory */
        return SVC_UNKNOWN ;
    }
40   savedFlow->connBlk = ipf->tcb ;
    savedFlow->connSeq = ipf->tcb->connSeq ;
    nessListAdd( sFlowList, savedFlow ) ;
    return MORE_MAGIC ;
}
45

/* If the response contains the single word "GET" or "SEND"
   then this is probably Napster data (upload/download) */
if ( ((ipf->header.dataLen == 3) && (dataStart[0] == 'G') &&
50       (dataStart[1] == 'E') &&
       (dataStart[2] == 'T')) ||

       ((ipf->header.dataLen == 4) && (dataStart[0] == 'S') &&
       (dataStart[1] == 'E') &&
       (dataStart[2] == 'N') &&
55       (dataStart[3] == 'D')) ) {

```

```

    curFlow.connBlk = ipf->tcb ;
    curFlow.connSeq = ipf->tcb->connSeq ;
    savedFlow = (struct napFlowListData *)
5         (nessListLookup( sFlowList, &curFlow )) ;

    /* If this flow is in the list of suspected Napster's
       then we probably have Napster Data traffic. */
    if (savedFlow) {
10         return SVC_NAP_DATA ;
    }
}

/* ----- */
15     }
}

/* If we make it to here then we did not recognize the traffic */
return SVC_UNKNOWN ;
20 }

/*
 * Parse the Napster Command flow for download requests.
 */
25 napParseFlow ( ) {

    char *dataStart ;    /* The start of the payload data unit */
    UINT16 cmdLength ;
    UINT16 cmdType ;

    whereStr("napParseFlow");
    debugBlurt("Parsing Napster-Cmd flow");

    if (!gNapCmdParsingEnabled) {
35         return FALSE ;
    }
    debugAssert(ipf->ucb || ipf->tcb);

    if (ipf->tcb) {
40
        /* Look for Napster commands */
        if (ipf->header.dataLen >= 4) {
            dataStart = ipf->header.dataStart ;

            /* If the first two bytes contain the length of the packet
               minus 4 bytes, then check for a reasonable type value */
            /* Note: The fields are not in network byte order, they are
               in little-endian form, so we can't use ntohs(). */
50
            cmdLength = (UINT16) (((UINT16)dataStart[1] << 8) & 0xff00) |
                        ((UINT16)dataStart[0] & 0x00ff) ;

            /* Check for a known 'type' */
55
            cmdType = (UINT16) (((UINT16)dataStart[3] << 8) & 0xff00) |
                    ((UINT16)dataStart[2] & 0x00ff) ;

```

```

    if (cmdLength == (ipf->header.dataLen - 4) ) {

        /* Currently the known command types are from 0 to 910,
        but I'm allowing some room to grow */
5       if (cmdType < 5000) {
            napParseCmd( ipf, cmdLength, cmdType, (dataStart + 4));
        }
    }
10    }
    /* Peek at this flow for as long as it exists. */
    return TRUE ;
}
15

/*
 * The first thing the Napster client does when it starts is make a connection
 * to the napster.com site where it is redirected to a server farm.
20 */
BOOL
napParseInitialConnection( UINT16 len, char *dataStart ) {

    /* The initial connection just contains an IP address and a port number.
    ie. 208.184.216.187:7777 terminated by an ascii newline (0x0A) and
    a null terminator (0x00) (maximum of 23 chars) */
25    IP_ADDR  serverAddr ;
    char*portString ;
    UINT16 portNumber ;
    BOOL  ret ;
30    int  i ;

    whereStr("napParseInitialConnection");

35    portString  = dataStart ;
    serverAddr  = 0 ;
    portNumber  = 0 ;
    ret        = FALSE ;

40    /* If it's too long then something is wrong, bail out */
    if (len > 25) {
        return FALSE ;
    }

45    for (i=0;i<len;i++) {
        if (dataStart[i] == ':') {
            dataStart[i] = '\0' ;
            serverAddr = inet_addr_in_host_order( dataStart ) ;
            dataStart[i] = ':' ;
50            portString = &(dataStart[i+1]) ;
        }
        if (dataStart[i] == '\n') {
            dataStart[i] = '\0' ;
            ret = napParsePort( portString, &portNumber ) ;
55            dataStart[i] = '\n' ;
            break ;
        }
    }
}

```

```

    }
}

5   if (serverAddr && ret && portNumber) {
    /* Remember this future flow */
    if (nessRememberRedirection(serverAddr, portNumber, TCP_PTCL,
        SVC_NAP_CMD)) {
        debugBlurt3("Remembering: 0x%x, %d, %d is Napster", serverAddr, portNumber,
10        SVC_NAP_CMD);
    }
    else {
        debugBlurt("DEBUG: failed to remember redirection");
    }
15   return TRUE ;
}
return FALSE ;
}

20 /*
   * Once the client establishes a connection to the database server, it
   * interacts via a message typed protocol. Each message begins with a
   * 2 byte length, then a 2 byte type, then the data. This function
25  * directs the data parsing based on the message type code.
   * Note: there can be more than one message in the TCP packet and the
   * messages can cross packet boundaries, but this code only looks at
   * the first message, and split messages are not processed.
   */
30  BOOL
napParseCmd( IPF_INFO_PTR ipf, UINT16 cmdLength, UINT16 cmdType, char *cmdDataStart)
{
    BOOL ret ;

35   whereStr("napParseCmd");

    switch (cmdType) {
        case 2:
        case 6:
40         debugBlurt1("Napster Login command: %d", cmdType );
        ret = napParseCmdLogin( ipf, cmdLength, cmdDataStart ) ;
        break ;
        case 216:
45         debugBlurt1("Napster Search command %d", cmdType );
        ret = napParseCmdNewAddr( cmdLength, cmdDataStart ) ;
        break ;
        case 204:
        case 501:
50         debugBlurt1("Napster Download/Upload command %d", cmdType );
        ret = napParseCmdNewAddr( cmdLength, cmdDataStart ) ;
        break ;
        case 300:
        case 703:
55         debugBlurt1("Napster Set Port command %d", cmdType );
        ret = napParseCmdNewPort( ipf, cmdLength, cmdDataStart ) ;
        break ;
    }
}

```

```

case 901:
    debugBlurt1("Napster Listen Test command %d", cmdType);
    ret = napParseCmdNewPort( ipf, cmdLength, cmdDataStart );
    break ;
5   default:
    ret = FALSE ;
}

10  return ret ;
}

/*
15  * The client login in message format is:
  * <username> <password> <port> "<client-info>" <link-type>
  */
BOOL
napParseCmdLogin( IPF_INFO_PTR ipf, UINT16 cmdLength, char *cmdDataStart ) {

20  char copyString[NAP_CMD_LOGIN_LENGTH];
  char      *copyStringKmem ;
  char      *marker ;
  char      *userString ;
  char      *passString ;
25  char      *portString ;
  UINT16    portNumber ;
  NessFlowPtr nf ;
  SIDE sside, dside;
  unsigned long saddr;

30  whereStr("napParseCmdLogin");

  copyStringKmem = NULL;
  if (cmdLength < NAP_CMD_LOGIN_LENGTH) {
35      memcpy(copyString, cmdDataStart, cmdLength);

      /* Must be null terminated for strtok_r */
      copyString[cmdLength] = 0;

40      userString = strtok_r( copyString, " ", &marker);
      passString = strtok_r( NULL, " ", &marker);
      portString = strtok_r( NULL, " ", &marker);
  } else {

45      /* cmdLength will be less than a packet in size */
      if (!(copyStringKmem = (char *)kmallocc( cmdLength + 1, M_NESS))) {

          /* Bail out because we're out of memory */
          return FALSE ;
50      }
      memcpy( copyStringKmem, cmdDataStart, cmdLength );

      /* Must be null terminated for strtok_r */
      copyStringKmem[cmdLength] = 0 ;
55      userString = strtok_r( copyStringKmem, " ", &marker);
      passString = strtok_r( NULL, " ", &marker);

```



```

    portString = strtok_r( NULL, " ", &marker );
}

/* Convert the port string into an integer */
5 if ( portString && napParsePort( portString, &portNumber )) {

    /* Now we can set some associative remembrance */
    /* If the port number is zero (which means that this client is
       behind a firewall) then there is no need to remember it */
10 if (portNumber) {
    nf = nessFlowFromIPF(ipf);
    ssize = bcb_direction(ipf->bcb);
    dside = (ssize ? 0 : 1);
    saddr = nf->conn.addr[dside];

15
    /* Remember this pending flow */
    if (nessRememberDependantRedirection(saddr, portNumber, TCP_PTCL,
        SVC_NAP_DATA, nf)) {
        debugBlurt3("Remembering Dependant Flow: 0x%x, %d, %d is Napster-Data",
20 saddr, portNumber, TCP_PTCL);
    }
    else {
        debugBlurt("DEBUG: failed to remember redirection");
    }
25 }
    if (copyStringKmem) kfree( copyStringKmem );
    return TRUE;
}
30 if (copyStringKmem) kfree( copyStringKmem );
    return FALSE;
}

/*
35 * The download messages are of the form:
* <username> <ip> <port> ...
*/
BOOL
napParseCmdNewAddr( UINT16 cmdLength, char *cmdDataStart ) {
40
    char copyString[NAP_CMD_NEWADDR_LENGTH];
    char *copyStringKmem;
    char *marker;
    char *userString;
45 char *addrString;
    char *portString;
    UINT16 portNumber;
    IP_ADDR peerAddr;

50 whereStr("napParseCmdNewAddr");

    copyStringKmem = NULL;

/*
55 * Look at the first NAP_CMD_NEWADDR_LENGTH bytes of the command.
* Considering that we're only interested in the first three fields

```

```

5  * of the command and that the <ip> and <port> pieces will only consume
   * a maximum of 17 bytes, including white space, then we only need
   * to account for a big username length. If it looks like the
   * username is so big that we may truncate the ip addr or port number,
   * then we must use kmalloc to correctly parse the command.
   */

   /* cmdLength will be less than a packet in size */
   memcpy(copyString, cmdDataStart, NAP_CMD_NEWADDR_LENGTH);

10  /* Must be null terminated for strtok_r */
   copyString[NAP_CMD_NEWADDR_LENGTH-1] = 0;
   userString = strtok_r(copyString, " ", &marker);

15  /* Check to see if the username was a resonable length */
   if ((marker - copyString) < (NAP_CMD_NEWADDR_LENGTH - 17)) {

       /* If username is okay, then get the remaining pieces */
       addrString = strtok_r(NULL, " ", &marker);
       portString = strtok_r(NULL, " ", &marker);

20  }

   /* Otherwise, it is an obnoxiously long username and we must use kmalloc */
   else {

25       /* cmdLength will be less than a packet in size */
       if (!(copyStringKmem = (char *)kmalloc(cmdLength + 1, M_NESS))) {

           /* Bail out because we're out of memory */
           return FALSE;

30       }
       memcpy(copyStringKmem, cmdDataStart, cmdLength);

       /* Must be null terminated for strtok_r */
       copyStringKmem[cmdLength] = 0;
       marker = NULL;
       userString = strtok_r(copyStringKmem, " ", &marker);
       addrString = strtok_r(NULL, " ", &marker);
       portString = strtok_r(NULL, " ", &marker);

35  }

40  /* Convert the port string into an integer */
   if (addrString && portString && napParsePort(portString, &portNumber)) {

45       /* Convert the address string to an IP_ADDR */
       peerAddr = (UINT32)strtol(addrString, (char **)0, 10);

       if (peerAddr != ULONG_MAX) {

50           peerAddr = rev4(peerAddr);

           /* Since this is a specific download command that will happen
              right now, we can use the regular remembrance */
           if (peerAddr && portNumber) {

55               /* Remember this future flow */
               if (nessRememberRedirection(peerAddr, portNumber, TCP_PTCL,

```

```

        SVC_NAP_DATA)) {
            debugBlurt3("Remembering: 0x%x, %d, %d is Napster-Data", peerAddr,
                portNumber, SVC_NAP_DATA);
        }
5         else {
            debugBlurt("DEBUG: failed to remember redirection");
        }
    }
10     if (copyStringKmem) kfree( copyStringKmem );
    return TRUE ;
}
if (copyStringKmem) kfree( copyStringKmem );
15 return FALSE ;
}

/*
20 * Some messages change or add port numbers and just contain the number:
* <port>
*/
BOOL
napParseCmdNewPort( IPF_INFO_PTR ipf, UINT16 cmdLength, char *cmdDataStart ) {
25     UINT16    portNumber ;
    NessFlowPtr nf ;
    SIDE sside, dside;
    unsigned long saddr;

30     whereStr("napParseCmdNewPort");

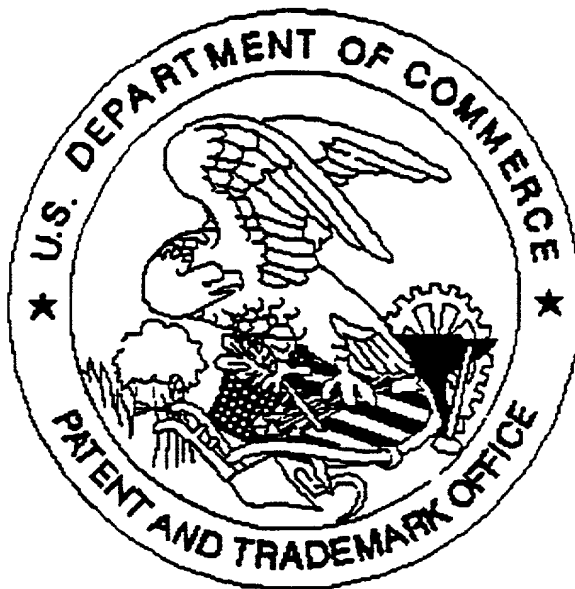
    /* Convert the port string into an integer */
    if ( napParsePort( cmdDataStart, &portNumber ) && portNumber ) {
35         /* Now we can set some associative remembrance */
        nf = nessFlowFromIPF(ipf) ;
        sside = bcb_direction(ipf->bcb) ;
        dside = (sside ? 0 : 1) ;
        saddr = nf->conn.addr[dside] ;
40
        debugBlurt2("PENDING FLOW : client 0x%x, port %d",
            saddr, portNumber );

        /* Remember this pending flow */
45         if (nessRememberDependantRedirection(saddr, portNumber, TCP_PTCL,
            SVC_NAP_DATA, nf)) {

            debugBlurt3("Remembering Dependant Flow: 0x%x, %d, %d is Napster-Data",
                saddr, portNumber, TCP_PTCL);
50         } else {
            debugBlurt("DEBUG: failed to remember redirection");
        }
        return TRUE ;
55     }
    return FALSE ;
}

```

United States Patent & Trademark Office  
Office of Initial Patent Examination – Scanning Division



Application deficiencies found during scanning:

☐ Page(s) \_\_\_\_\_ of \_\_\_\_\_ were not present  
for scanning. (Document title)

☐ Page(s) \_\_\_\_\_ of \_\_\_\_\_ were not present  
for scanning. (Document title)

*Specification contains appendix in pages 15-25.*

☐ *Scanned copy is best available.*